Entwurfsbeschreibung

Gliederung

1 Einleitung

1.1 Verwendungszweck

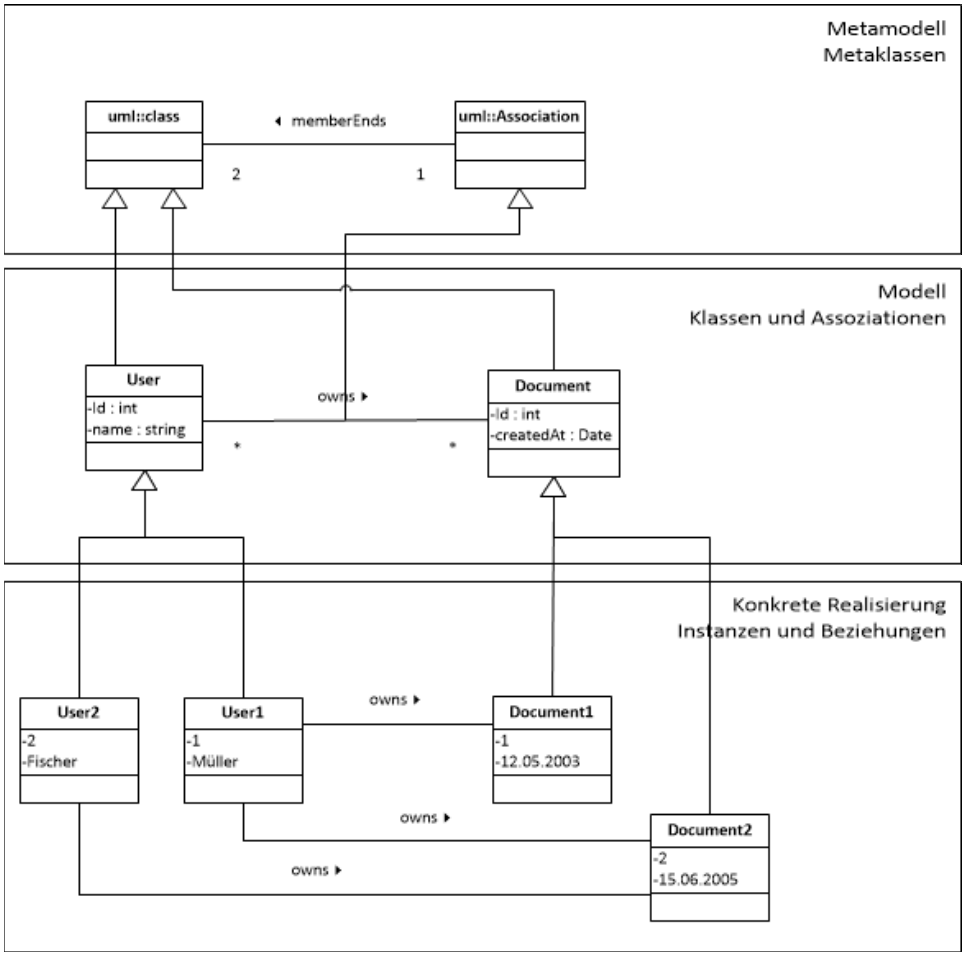
Ziel der Projektes ist es eine Software zu entwickeln, die aus einem annotierten UML Diagramm nach UML 2.0 Standart eine Webanwendung (Servlet) mit Datenbankanbindung erzeugt. Dabei sollen alle im UML Diagramm enthaltenen Informationen (Assoziationen, Vererbung, Attribute, Multiplizitäten) auf eine Datenbank abgebildet werden. Desweiteren soll in der Webanwendung eine Nutzereingabe unter Einhaltung der im UML spezifizierten Restriktionen möglich sein, dazu gehört sowohl das Anlegen von Assoziationen zu Klassen als auch das Anlegen von Attributen eines bestimmten Datentyps.

1.2 Allgemeine Bemerkungen zum Projekt

Größte Herrausforderung bei der Umsetzung dieses Projektes war die Generierung validen Codes aus einem beliebigen UML Diagramm. Es musste in jedem Entwicklungsschritt sichergestellt werden das Änderungen nicht nur für eine Klasse sondern immer für alle nach UML 2.0 Standart möglichen Formen einer Klasse umgesetzt werden. Auf die Besonderheiten einer solchen Modellgetriebenen Softwarearchitektur wird nochmal in Abschnitt 1.2 eingegangen, auch liegt darin die strikte Aufteilung des Projektes in 3 Teile begründet. Diese sind Generator, Webanwendung und Datenbankanwendung. Auf diese Trennung wird auch während der Entwicklung wertgelegt, d.h. es wird eine neue Funktion wird zuerst für eine spezielle Klasse auf Seiten der Webanwendung implementiert, dann die korrekte Einbindung in die Datenbank sichergestellt und danach erst auf eine allgemeine Klasse abgebildet und in den Ablauf des Generators eingebunden. Nach Vorbild dieser Entwicklungsschritte ist auch diese Entwurfsbeschreibung strukturiert. Also wird zuerst auf die Funktionen und Technologien der Webanwendung (Abschnitt 2), dann und auf die der Datenbankanwendung (Abschnitt 3) und zuletzt erst auf die des Generators (Abschnitt 4) eingegangen.
Der eigentliche Ablauf der Software steht dem jedoch entgegen und wird daher im Abschnitt 1.3 genauer beschrieben.

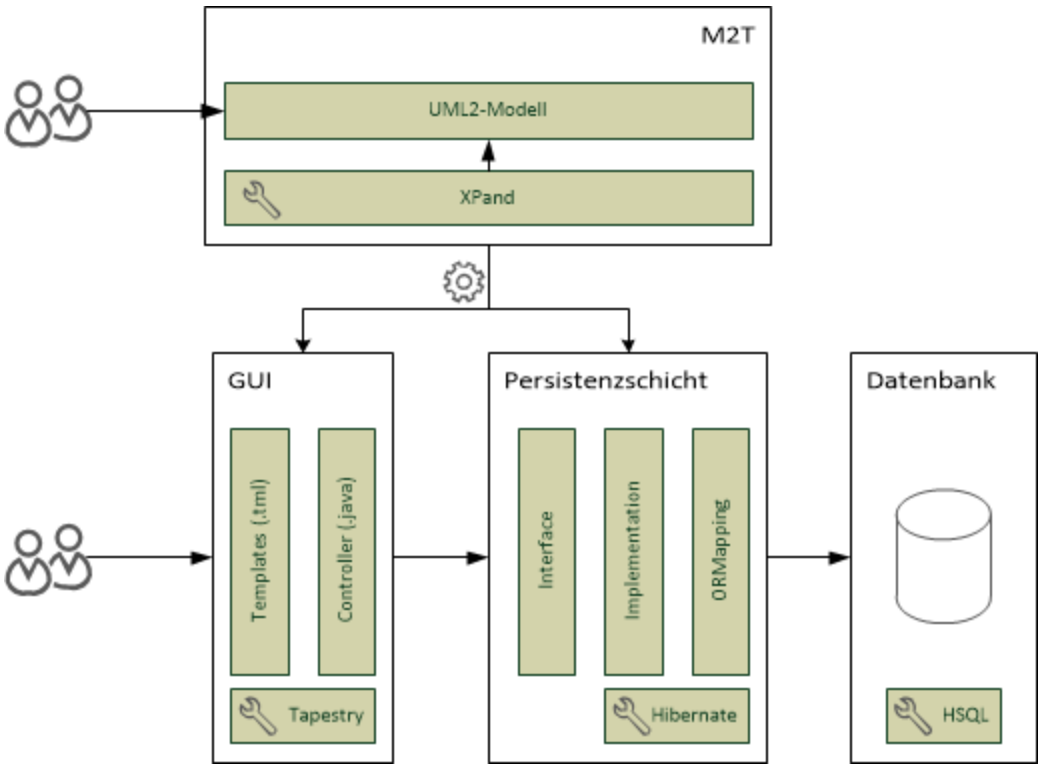
1.3 Modellgetriebene Softwarearchitektur (MDA - ModelDrivenArchitecture)

Da die Struktur des zu entwickelnden dynamischen Servlets daher fast vollständig von der Eingabe des Nutzers (UML-Diagramm) abhängt, erscheint es sinnvoll die Softwarearchitektur auf Grundlage des modellgetriebenen Ansatzes zu erstellen.
Dabei ist eine saubere Abstraktion aller umzusetzenden Features zu beachten, im Gegensatz zu herkömmlichen Softwarearchitekturen ist MDA prinzipiell in der Ebene des Metamodells zu entwerfen, da uns das Modell nicht bekannt ist (Eingabe).



In unserem Fall entwickeln wir ein Tool zur Transformation von Modelldaten in funktionsfähigen Quellcode (M2T-ModelToText).

1.4 Projektarchitektur



Das Servlet ist in 2 Schichten aufgeteilt:
1. Schicht zur Realisierung der Benutzeroberfläche, diese wird mittels des Tapestry-Frameworks umgesetzt (siehe Kapitel 2).
2. Schicht zur Realisierung der Persistenz, diese wird mittels des Hibernate Frameworks umgesetzt. (Kapitel 3)

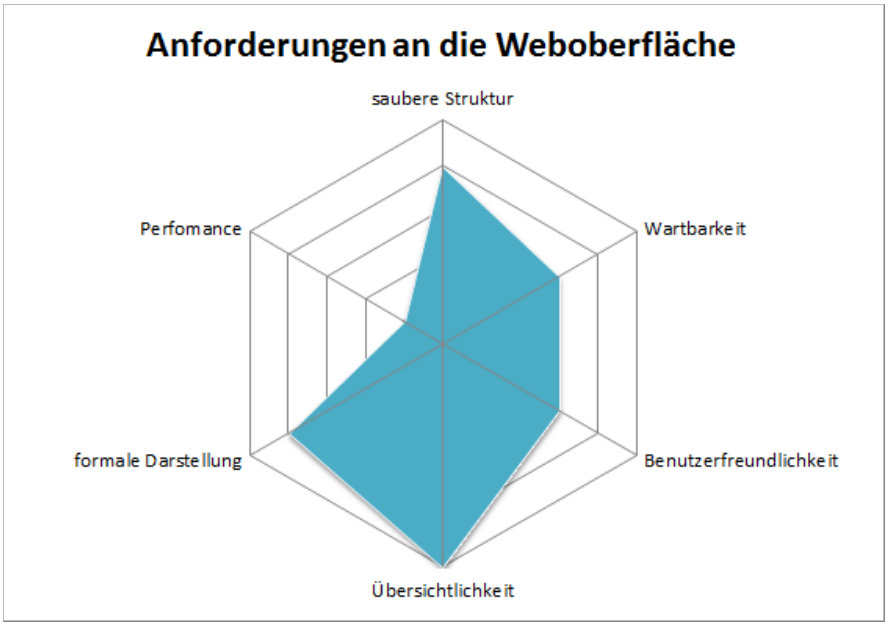
Zur Realisierung des Generierungsprozesses (M2T) benutzen wir XPand (Kapitel4).

Aus der Obigen Grafik lässt sich auch der Ablauf der Software entnehmen. Im ersten Schritt gibt der Nutzer ein UML Diagramm ein, daraus wird dann vom Codegenerator das Servlet generiert. Danach kann der Nutzer über die Webanwendung eingaben tätigen, diese werden dann von der Persistenzschicht validiert und auf der Datenbank gespeichert.

2 Servlet - Weboberfläche

2.1 Anforderungen

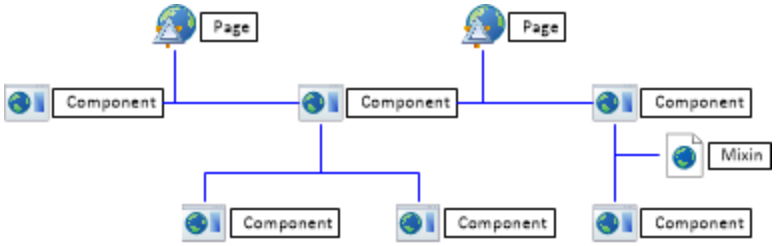
Die Weboberfläche soll die Lebenslaufakte in der vom User festgelegten Struktur darstellen und es ermöglichen Objekte dieser Struktur anzulegen, anzuzeigen, bearbeiten und löschen zu können. Dabei muss die Oberfläche offenbar sehr dynamisch ausfallen und dennoch strukturiert gestaltet sein, da sie zur Verwaltung der Objekte dient und keinesfalls zu unübersichtlich werden darf. Dabei ist eine formal saubere Darstellung der Objekte und eine gute Struktur wichtig. Die Perfomance spielt eine untergeordnete Rolle. Im Hinblick auf weitere Entwicklung und eine gute Teamarbeit ist auch auf Wartbarkeit des entwickelten, sowie des generierten Codes zu achten.



2.2 Tapestry Framework

Tapestry ist ein OpenSource Web-Framework für Java, mit welcher man ein Sevlet auf Basis der Java Servlet API erstellen kann. Wir verwenden Tapestry 5.3. Tapestry-Seiten und Tapestry-Komponenten setzen sich dabei aus 2 verschiedenen Bestandteilen zusammen:
1. Template-Datei (.tml), die sowohl XHTML-Tags als auch Tapestry-spezifische Tags zum Aufbau der Seite und deren Darstellung beinhaltet
2. Controller-Datei (.java, POJO's), die die Funktionalität der Seite realisiert
3. optional eine Properties-Datei (.properties), die bestimmte Eigenschaften der Seite / Komponente beschreibt, inkl. eines Message-Catalogs zum Auslagern von Nachrichten zur besseren Lokalisierung der Software.

Dabei werden Seiten aus verschiedenen Komponenten zusammengesetzt, diese Komponenten können andere Komponenten beeinhalten und durch sog. Mixins angepasst werden. Komponenten können aus der Tapestry-Standardbibliothek, aus importierten Bibliotheken stammen oder auch selbst entwickelt werden. Dies gestattet eine saubere und objektorientierte Struktur der Website.



Tapestry orientiert sich zudem an Annotations und NamingConventions für Konfigurationseinstellungen anstatt über zusätzliche Dateien. Das macht den Code lesbarer und wartbarer. Auch ist die Aufteilung in Oberflächenaufbau (.tml) und Funktionalität (.java) für Teamarbeit gut geeignet, da HTML- und Java-Entwickler parallel an einer Seite/Komponente arbeiten können. Zudem bietet Tapestry guten Support für unser Backend (Hibernate).

2.3 Tapestry-Projekt-Struktur

Das Websiteprojekt besitzt eine feste Ordnerstruktur:

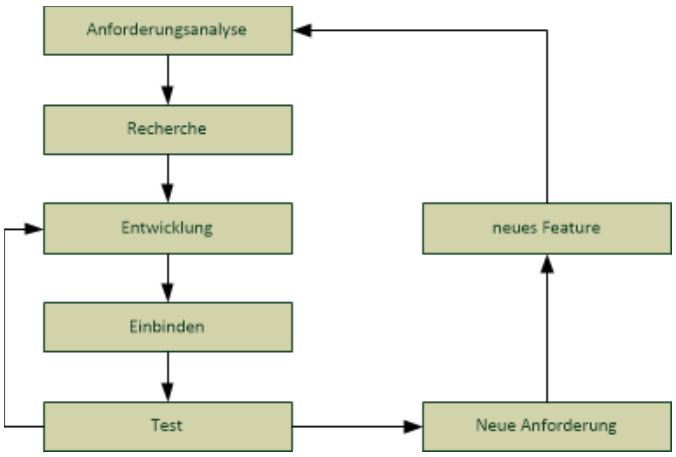
- Der Ordner src/main/java enthält Java-Quellcode Dateien
- src/main/resources enthält weitere Ressourcen wie Templates(.tml)
- src/main/webapp enthält spezielle Webanwendungs-Dateien . Beispiel dafür ist die Layout.css, die als Designgrundlage für alle Seiten dient.

Das Package la.website enthält die wichtigsten Dateien:

- la.website.pages enthält die Tapestry Seiten
- la.website.components enthält selbst erstellte Tapestry Komponenten
- la.website.entities enthält die Hibernate-Entities. (siehe Kapitel 3)
- la.website.data enthält Klassen, die indirekt benutzt werden, wie Enumerations oder DAO's
- la.website.services enthält Service-Module. Dies sind verschiedene Module zum Einbinden von Bibliotheken, zur Konfiguration der Webanwendung etc.

2.4 Komponenten und Seiten

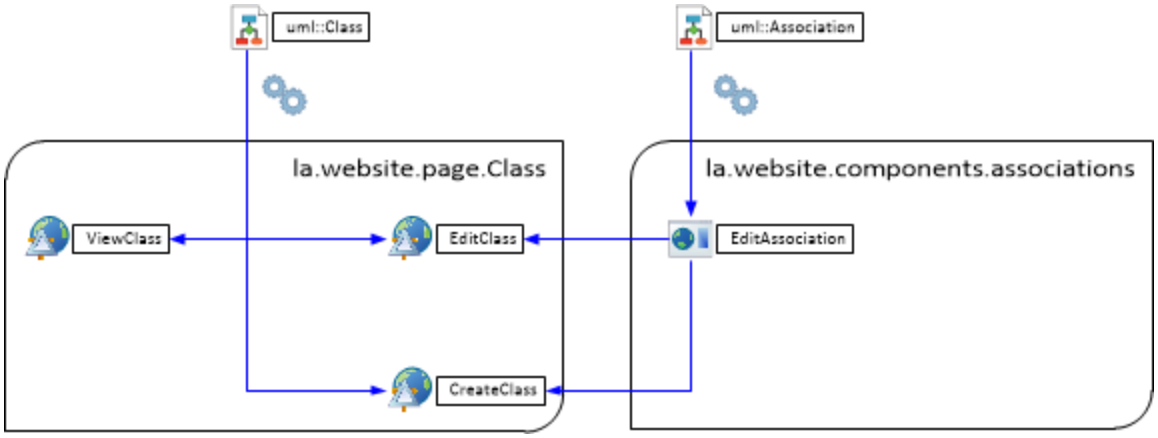
Die Grundlage für jedes Tapestry-Projekt bilden Komponenten, die zu den verschiedenen Seiten der Website zusammengesetzt werden. Eine komponentenbasierte Entwicklung ermöglicht einen an der Struktur der zu verwaltenden Objekte (UML-Diagramm) orientierten Aufbau der Website und eine inkrementelle Entwicklung der Website, die dadurch schrittweise um Komponenten für bestimmte Funktionalitäten erweitert werden kann.



Komponenten sind demnach in Hinsicht auf bestimmte Funktionalitäten für bestimmte Objekte zu entwickeln. Auch ermöglicht uns diese Art der Entwicklung auf bereits bestehende Bibliotheken zurückzugreifen.

Tapestry Komponenten werden durch einen normalen HTML-Tag beschrieben und interpretiert. Zudem bietet Tapestry einige Annotations, beispielsweise ist es möglich mittels der @Property Annotation Getter und Setter für ein Objekt einzubinden ohne diese extra erstellen zu müssen. Dies gestaltet die Dateien übersichtlicher und verringert den Programmieraufwand. Zudem ergänzen sie den Code in einer lesbaren Weise.

Für jede Klasse aus dem eingegebenen UML-Modell wird beim Generieren ein Package in la.website.pages angelegt und dort werden Seiten zum Anlegen, Bearbeiten und Anzeigen einer Instanz dieser Klasse generiert. Zusätzlich wird für jede Assoziation eine Komponente zum Bearbeiten der Assoziationspartner generiert (im Package la.website.components.associations), die in der Anlegen- und Bearbeiten - Seite eingebunden wird (Assoziationen werden im Kontext von Instanzen editiert), siehe Grafik:



Eine detaillierte Anleitung zum Erstellen, Bearbeiten und Löschen von Klassen, Assoziationen und Attributen finden sie im Benutzerhandbuch.

2.5 Seiten binden an Objekte - PageActivationContext

Viele Seiten machen nur im Kontext zu bestimmten Objekten Sinn. So ist bspw. der Bearbeiten Seite ein Objekt zu übergeben, welches bearbeitet werden soll. Dies geschieht über den PageActivationContext. So wird beim Aufruf der Bearbeiten-Seite für eine Instanz der Klasse “User” die onActivate-Methode aufgerufen:

```
public class EditUser {

	@Persist("entity")
	@Property
	private User UserObj;

	[...]

	void onActivate(User obj) {
		UserObj = obj;
	}
}
```

Jeder Aufruf der Bearbeiten Seite muss im `context` einer Instanz der Klasse User geschehen:

```
<t:pagelink page="User/Edit" context="UserObj">
	${message:edit-label}
</t:pagelink>
```

2.6 Konsistenz wahren - Validierung von Eingaben

Mit der @Validate Annotation über einem Attribut lassen sich Nutzereingaben auf der Website überpüfen. Möglich ist dabei festzulegen ob etwas zwingend notwendig ist, ob es einen minimalen oder maximalen Wert haben muss, oder ob eine Zeichenkette eine bestimmten Muster entsprechen muss.

Auch lassen sich Eingaben in Formularen oder in Komponenten in einem Formular-Kontext (wie die AssociationsEdit-Komponente) über Event-Methoden validieren, die Tapestry triggert, wenn das Formular abgeschickt wird (Im Falle von Komponenten bedarf es noch einen kleinen Workaround um die Validierung auch an die Komponente weiter zu delegieren). Mit einem Error-Tracker werden alle Fehler, die auftreten, ermittelt und dem User danach mit einer entsprechenden Fehlermeldung angezeigt.
Beispiel:

```
private void processSubmission() {
            // Validierung.
            if (editObject.getMemberOfUser().size() <
                  editObject.MUL_MEMBEROF_USER_LOWER)
            {
                  //Lower Value (Multiziplitaet) unterschritten
                  tracker.recordError(this, messages.format("lowerValueError",
                              editObject.MUL_MEMBEROF_USER_LOWER));
            }
            if (editObject.getMemberOfUser().size() >
                  editObject.MUL_MEMBEROF_USER_UPPER)
            {
                  //Upper Value (Multiziplitaet) ueberschritten
                  tracker.recordError(this, messages.format("upperValueError",
                              editObject.MUL_MEMBEROF_USER_UPPER));
            }
      }
```

3 Objektrelationales Datenbanksystem

3.1 Objektrelationales Mapping - Das Hibernate Framework

Das Mapping der zugehörigen Objekte auf die Datenbank erfolgt dabei durch Hibernate. Die genaue Spezifikation des Mappings erfolgt in einer Mapping-Datei. Im src-Verzeichnis steht die *hibernate.cfg.xml* Datei und sie sieht folgendermaßen aus.

```
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

   <session-factory>
      <property name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver
      </property>
      <property name="hibernate.connection.url">
            jdbc:hsqldb:./target/db/swpla;shutdown=true
      </property>
      <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
      </property>
      <property name="hibernate.connection.username">sa</property>
      <property name="hibernate.connection.password"></property>
      <property name="hbm2ddl.auto">update</property>
      <property name="hibernate.show_sql">true</property>
      <property name="hibernate.format_sql">true</property>
   </session-factory>

</hibernate-configuration>
```

Die Konfiguration enthält zu Beginn die Datenbankverbindung. Anschließend legen wir mit der Einstellung *dialect* fest, in welchen SQL - Dialekt Hibernate Abfrage übersetzt. Unser Mapping erfolgt durch *Annotations.*
Mit dem Mapping legen wir fest, welcher Tabelle und welcher Spalte ein Attribut einer Klasse zugeordnet ist. *Annotations* kennzeichnen dabei Elemente (wie Klassen, Attribute, Assoziationsattribute) und weisen ihnen bestimmte Konfigurationseinstellungen für das Mapping zu. Dies ersetzt die sonst recht aufwendige Konfigurationseinstellung durch extrene Dateien.

3.2 Wichtige Annotations

3.2.1 @Entity-Annotation

Mit der *@Entity-Annotation* wird eine Klasse als persistente Entität markiert. Dabei kann mittels @Table u.a. der Tabellenname und mit @Coloumn u.a. der Spaltenname für ein bestimmtes Attribut angepasst werden (ORM-Konfiguration). Das ist Wichtig, da die Namen für Klassen beliebig sein können und sich unter Umständen mit SQL-Befehlen überlagern (Group, Order).

3.2.2 @CommitAfter Annotation

Mittels *@CommitAfter* werden Methoden versehen, die Transaktionen realisieren. Diese Art der Kennzeichnung von Transaktionsmethoden bietet sich an, da Transaktionen nicht manuell erstellt oder commited werden müssen. Beim Auftreten einer Exception wird ebenfalls ein automatisches Rollback vorgenommen.

3.3 Datenbanksystem - HSQLDB

Als Datenbank verwenden wir HSQLDB. Diese Datenbank lässt sich ganz einfach installieren und konfigurieren. Außerdem erlaubt es den Betrieb der Datenbank im Hauptspeicher (In-Memory-DB) ohne Sicherung auf der Festplatte. Sie ist daher zum Testen und Ausprobieren sehr gut geeignet.

4 Codegenerator - XPand

4.1 Xpand

4.1.1 Funktionsweise von XPand

Technologische Grundlage des Generators ist die Templatesprache Xpand, also eine Sprache, die dazu dient, aus Vorlagen mit Platzhaltern (Templates) fertigen Code zu erzeugen. Diese deklarative Herangehensweise hat den Vorteil, dass der Fokus bei der Implementierung des Generators auf der Struktur des zu erstellenden Textes liegt und nicht bei der Realisierung technischer Details bezüglich Textaufbau (z.B. das Auffinden der dynamisch zu verändernden Textstellen), Dateizugriff oder ähnlichen Dingen.
Das gesamte Xpand-Projekt besteht aus Modellen, aus denen Code generiert werden soll (siehe dazu 4.2), Templates, die die Vorlage für den zu genierenden Code darstellen (siehe 4.5.4), Checking-Constraints, um Bedingungen zu überprüfen, die erfüllt sein müssen damit sinnvoller Code generiert werden kann (siehe 4.5.2), Extensions, die eine bessere Strukturierung des Projektes ermöglichen (siehe 4.5.3) und eine Workflow-Datei, die den gesamten Generierungsprozess steuert (siehe 4.5.1). Mehr über Xpand finden Sie auf dieser Seite.

4.2 Eingabemodell

Das zu verwendende UML-Modell muss in Form einer XML/XMI-Datei nach OMG-Standart vorliegen. Für das Modell selbst ist die Einhaltung des UML 2.0-Standart erforderlich.
Weiterhin ist zu beachten, dass ein primitives Attribut immer eine konkrete obere Grenze haben muss, andernfalls (für variable Grenzen) muss eine Assoziation verwendet werden.
Umgesetzt werden aus dem UML-Modell die Klassen mit ihren primitiven Attributen (soweit diese im verwendeten UML-Profile angegeben sind), Assoziationen (inklusive Multiplizitäten) und Vererbung, außerdem auch Enumerationen sowohl als Java-Quelltext als auch in Form einer Abbildung auf die objektrelationale Datenbank (siehe 3). Des Weiteren werden die aus diesen genannten Informationen gegebenen Einschränkungen (z.B. Multiplizitätsgrenzen bei Assoziationen) beachtet, wenn der Benutzer über die Webanwendung Objekte hinzufügt oder bearbeitet, sodass keine zum Modell inkonsistenten Instanzen erstehen können.
Zur Erstellung dieses Modells kann Eclipse mit der Erweiterung Topcased verwendet werden. Mit Hilfe des Bausteinprinzips kann man ein Papyrus Modell erstellen.

4.3 Allgemeiner Ablauf des Generators

Der gesamte Generierungsvorgang besteht aus den folgenden Schritten:
Zuerst werden die Einstellungen aus der "generator.properties"-Datei geladen, dann wird das UML-Modell (entsprechend den in 4.2 beschriebenen Anforderungen) eingelesen. Als nächstes werden die Checkings-Constraints (mehr dazu in 4.5.3) und Metamodelle eingebunden.
Zuletzt wird das initiale Template (welches in unserem Fall “Root” heißt, näheres dazu unter 4.5.4) geladen und daraus der gesamte Code generiert (wobei das initiale Template weitere Templates aufrufen kann, siehe dazu 4.5.4), wobei vorher die Ordner bereinigt werden.

4.5 Umsetzung des Codegenerators mit Xpand

4.5.1 Workflow

Die Workflow-Datei ist eine XML-Datei, die den gesamten Generierungsvorgang steuert und besteht aus folgenden Bestandteilen in Form von Komponenten:

- Der eigentliche Generator, bestehend aus
    - Festlegung der Kodierung für die zu erstellenden Dateien
    - Einbindung der Metamodelle
    - Zu verwendendes Template
- Zu ladendes Modell
- Angabe, welche Verzeichnisse bereinigt werden sollen
- Zu verwendende Checking-Constraints (dazu mehr im nächsten Abschnitt)

Die dafür notwendigen Einstellungen, wie die zu verwendenden Verzeichnisse für die Generierung sowie das zu ladende UML-Modell werden in einer "properties"-Datei gespeichert.

4.5.2 Extensions

Extensions sind Dateien, in denen oft verwendete Funktionen ausgelagert werden. Sie bieten uns den Vorteil, dass Projekt besser strukturieren zu können können.
Dafür werden die Funktionen in einer Extensions-Datei definiert und können dann in einem beliebigen Template eingebunden und aufgerufen werden. Eine Anpassung der Workflow-Datei ist dabei nicht notwendig.
Weitere allgemeine Informationen zum Thema Extensions finden Sie auf dieser Seite.
In diesem Generator werden drei Extensions verwendet:
Properties.ext enthält Funktionen, die einfach nur Pfade von Ordnern für die zu speichernden Dateien zurückliefert.
Vererbung.ext liefert Funktionen zur Abfrage der Ober- und Klassen.
Associations.ext liefert Funktionen, die Assoziationen liefern bzw. assoziierte Klassen und deren

Kardinalitätseigenschaften sowie Navigierbarkeit überprüfen.

4.5.3 Checks

Checking-Constraints sind Dateien, in denen Bedingungen angegeben werden, die erfüllt sein müssen, damit die Generierung ausgeführt werden kann. Dadurch kann gewährleistet werden, dass nur solche Modelle geladen werden, die sich für diese Templates eignen.
Dazu wird ein Überprüfungsbereich angegeben (z.B. Modell, Klasse, Attribute), in dem die Bedingung überprüft werden soll, dann die auszugebende Fehlermeldung und dann die Bedingung in Xpand-Syntax. Zu beachten ist, dass die verwendeten Checking-Costraints in der Workflow-Datei angegeben werden müssen, damit sie ausgeführt werden. Wenn dann der Workflow gestartet wird, werden die Checking-Constraints von der Workflow-Engine überprüft und der Generierungsvorgang nur dann ausgeführt, wenn alle Bedingungen erfüllt sind.
Weitere allgemeine Informationen zum Thema Checkings-Constraints finden Sie auf dieser Seite.
Dies wird in den Checking-Constraints “leerePackages.chk” und “ohneAttribute.chk” dazu genutzt, um Klassen ohne Attribute oder mit leeren Packages zu erkennen und abzulehnen. Dazu wird einfach die Liste aller vorhandenen Attribute bzw. der vorhanden Elemente und der Packages der Klasse auf Leerheit überprüft.
Weitere Anwendungen sind das Überprüfen, ob eine (in Java unzulässige) Mehrfachvererbung im Modell auftritt sowie eine Einschränkung der verfügbaren primitiven Datentypen. Für die Mehrfachvererbung wird in “mehrfachVererbung.chk” dabei die Größe der Menge aller Basisklassen untersucht, die Extension "Vererbungen.ext" liefert dafür entsprechende Funktionen.
Um die Einschränkung der Datentypen zu realisieren, werden alle Attribute daraufhin überprüft, ob im Modell eine Klasse oder eine Enumeration existiert, die dem Typ des Attributs entspricht, oder, falls es sich um ein primitives Attribut handelt, wird in einem speziell dafür erstellten UML-Profile nach dem Typen des Attributs gesucht. Dieses UML-Profile, in dem also festgelegt wird, welche primitiven Attribute erlaubt sind, muss dazu in Xpand als Metamodell eingebunden werden.
Die Einbindung des Metamodells findet dabei in der bereits erwähnten Workflow-Datei statt.

4.5.4 Templates

Die eigentliche Codegeneration erfolgt in den XPand Templates (src/templates). Wenn ein Template Extensions nutzt bezieht es sich immer auf eine gleichnamige .ext Datei (in src/extensions). Besonders herrauszustellen ist hier jedeoch die properties.ext diese wird von allen Templates genutzt und ermöglicht Änderungen an der für Tapestry benötigen Ordernstruktur (siehe 2.3) ohne Änderungen an den Tempaltes vorzunehmen.
Auch gibt es Templates die eine gewisse Sonderstellung haben und auf die hier expliziet

eingegangen werden soll, diese sind:

Root.xpt	Zentraler Aufruf aller anderen Templates, steurt den Ablauf der anderen Templates
Once.xpt	Erzeugt die für die Webseite/Datenbank benötigen nicht dynamischen Teile
Primitivs.xpt	Mapping der primitiven Datentypen auf Javadatentypen sowie zugehörige Tapestry Validierung

Die anderen Templates sind alle nach einem ähnlichen Schema aufgebaut und haben den gleichen Ablauf, daher soll dieser im Folgenden nur Anhand eines Beispiels genau erklärt werden. Diese Templates sind:

Creat.xpt	Erzeugt alle nötigen Java und Tml Dateien zum Erzeugen einer Klasse
View.xpt	Erzeugt alle nötigen Java und Tml Dateien zum Anzeigen einer Klasse
Edit.xpt	Erzeugt alle nötigen Java und Tml Dateien zum Bearbeiten einer Klasse
Entites.xpt	Erzeugt alle nötigen Java Dateien zum Abbilden einer Klasse auf die Datenbank
Enumerations.xpt	Erzeugt alle nötigen Java und Tml Dateien zum benutzen einer Enumeration
AssociationsEdit.xpt	Erzeugt alle nötigen Java und Tml Dateien zum Hinzufügen einer Association zu einer Klasse
RessourcePages.xpt	Erzeugt alle nötigen Tml Dateien zur Darstellung aller Seiten der Website
JavaPages.xpt	Erzeugt alle nötigen Java Dateien zur Darstellung aller Seiten der Website

Ausgangspunkt bildet wie oben erwähnt das Root Template. Das Besondere ist das es als einziges Template direkt aus dem Workflow aufgerufen wird, es bekommt dabei das eingelesene UML Diagramm (Modell) mit allen Metamodells (u.a. für die primitiven Datentypen)

übergeben. Es ruft nun alle weiteren Templates nacheinander auf, und übergibt ihnen das eingelesene Modell. Die Reihenfolge des Aufrufs der anderen Templates spielt dabei keine Rolle. Dies wird dadurch gewährleistet, dass keine zwei Templates eine gleichnamige Datei generieren, darauf ist auch beim hinzufügen eventuell neuer Templates zu achten. Ansonsten ist das Einbinden neuer Templates durch einen Aufruf in Root leicht umzusetzen.

Im Template Once werden in alle nicht dynamischen Teile des Projekts erzeugt, dh alle Teile die nicht von dem eingegeben UML Diagramm abhängen. Insbesondere auch die Config Files für die Hibernate/HSQLDB (siehe 3.1) und das Layout der Webseite. Außerdem noch die Startseite und die Seite zu Kontaktinformationen.

Das Primitivs Template mappet die im primitivs.profile.uml (src/profiles) festgelgten Datentypen auf entsprechende Javadatentypen mit einer angepassten Tapestry @Validate Annotation (siehe 2.6). Durch diese Annotation wird für jeden Datentyp gesichert das nur eine valide Eingabe das Nutzers akzeptiert wird. Außerdem kommen noch die benötigten Hibernate Annotations (siehe 3.2) zu jedem Attribute hinzu. Unterstütze Datentypen sind:

String, Boolean, SignedInteger, Integer, UnsignedInteger, SignedLong, Long, UnsignedLong, Float, Double, DateTime, Time, Date, Duration, Byte, UID

UID ist der einzige Datentyp der nicht vom Nutzer eingegeben wird, sonder beim erstellen eines Objektes automatisch erstellt wird. Er soll als Global eindeutige Identifizierungsnummer eines Objektes gelten, und durch die automatische Erstellung sind Doppelungen ausgeschlossen. Zur Generierung wird dabei auf java.util.UUID zurückgegriffen.
Alle anderen Werte der Datentypen müssen vom Nutzer beim Erstellen oder Bearbeiten eines Objektes gesetzt werden. Dabei wird berücksichtigt ob eine Attribute nach Vorgabe das UML Diagramms zwingend notwendig ist oder nicht. Weiterhin werden auch Multiplizitäten primitiver Attribute berücksichtigt, wobei immer eine konkrete obere Grenze anzugeben ist (wenn eine Klasse beliebig viele Attribute haben soll ist dies als Beziehung umzusetzen, siehe 4.2).
Wichtig ist an dieser Stelle nocheinmal herrauzustellen das Attribute die sich nicht auf eine Assoziation beziehen und keinem der obigen Datentypen entsprechen nicht umgesetzt werde.
Weitere Besonderheit das Primitivs Templates ist das es nicht von Root, sonder erst aus dem Entities Template aufgerufen wird. Ihm wird dabei nicht das UML Model übergeben, sondern eine Liste mit den für das Mapping benötigten Informationen (u.a. mit dem Attribut).

Der allgemeine Aufbau eines XPand Templates sei kurz anhand des Templats Entities.xpt erklärt. Zunächst werden benötigte Extensions und Templates eingebunden. Danach folgt ein Define-Block der die Funktionalität dieses Templates erklärt, er muss immer jeweils den Namen das Templats tragen da genau dieser Teil ausgeführt wird wenn man ein Template aufruft. Mit dem Expand Statment werden nun die einzelnen Funktionen eines Templates aufgerufen. Diese Funktionen sind wieder mit einem Define-Block zu definieren. Dabei wird meist im Gegensatz zum eines Templates nichtmehr das ganze Modell übergeben sondern ein Bestimmter Teil des Modell, im Fall der Entities.xpt werden alle Klassen das Modells übergeben. Dabei wird dann

also auch die Funktion für jede Klasse ausgeführt. Bei allen der hier zu beschreibenden Templates wird nun eine Datei erstellt. Der Pfad wird dabei wie oben beschrieben durch die properties.ext festgelegt, und der Name bildet eine Kombination aus Templatename und Name des übergebenen Objekts. Nun wird der benötigte Quellcode in die Datei geschrieben. Im Fall der Entites.xpt Java Quellcode. Der zu schreibene Quellcode unterscheidet sich natürlich ja nach Funktionalität das Templates (siehe Tabelle oben).
So können auch mehrere Dateien pro Template erzeugt werden, und insgesamt ergibt sich damit nach ausführen aller Templates eine vollständiges Servlet wie in Abschnitt 2 beschrieben.

5 Entwicklungswerkzeuge

5.1 Entwicklungsumgebung - Eclipse IDE

Eclipse ist die Standard-Open Source Java-Entwicklungsumgebung, die durch die Möglichkeit unterschiedlichste Plugins einzubinden, die Chance bietet die Entwicklungsumgebung auf das Projekt anzupassen, z.B. durch die Implementierung von EGit oder Maven in Eclipse. Die größten Vorteile bilden die Plattformunabhängigkeit und die große Anpassungsfähigkeit des Systems an verschiedenste Anforderungen.

5.2 Git

Git ist ein Versionsverwaltungssystem, welches eine dezentrale Kontrolle und Aktualisierung des Projektes ermöglicht und jedem überall, zu jeder Zeit die Chance bietet auf das Projekt zuzugreifen und zu bearbeiten. Dabei ermöglicht es durch Versionierung und Branches paralleles Arbeiten an einer Schnittstelle und gleichzeitiges Arbeiten an verschiedenen Stellen.

5.3 Maven

Maven ist ein deklaratives Build Management System. Das heißt, es wird lediglich der Inhalt des Projekts beschrieben, nicht die Struktur oder die Abläufe, die zur Kompilierung und Veröffentlichung notwendig sind. Die Philosophie hinter Maven heißt Konvention vor Konfiguration
– Strukturen müssen nicht definiert werden, sondern sind vorgegeben. So wie die Projekt- und Verzeichnisstruktur ist auch die Reihenfolge der Arbeitsschritte vorgegeben, die Maven ausführt, um ein Projekt zu bauen.

6 Testkonzept

6.1 Allgemeines

Heutzutage ist die Anzahl der Fehler bei der Entwicklung einer komplexen Software sehr hoch. Aus diesem Grund ist ein Testkonzept für den Entwurf guter Software notwendig, denn das ermöglicht Fehler aus einem Softwaresystem zu verringern. Des Weiteren ist es wichtig während der Entwicklung einer Software kontinuierlich zu testen. Jedes Projekt soll am besten in Teilprojekte geteilt werden. Dies erleichtert die Suche nach möglichen Fehlern und deren Korrektur.

6.2 Testframework - JUnit

Die Tests von Klassen und ihre Funktionalität sollen mit Hilfe eines Test-Frameworks durchgeführt werden. Das Test-Framework *JUnit* ermöglicht dem Programmierer, einzelne Units, Klassen oder Methoden, von Java-Programmen zu testen.

JUnit bietet unter anderem verschiedene Methoden(Wie z.B., assertTrue(), assertFalse(), assertEquals()), um leicht Bedingungen im Code nachzuprüfen.

Um eine Klasse zu testen, werden in der Regel folgende Methoden erstellt und mit einer entsprechenden *Annotation* versehen:

- eine *Setup*-Methode, die alle nötigen Objekte anlegt und auf den für den Test nötigen Ausgangszustand bringt.
- eine Reihe von Tests, die die eigentlichen Test beinhalten.
- eine *Teardown*-Methode, um aufzuräumen.

Nachdem man die nötigen Methoden zum Testen erstellt hat, ist es dann möglich die Tests automatisch ablaufen zu lassen, um sich die Ergebnisse und Ursachen im Falle eines Fehlschlags anzeigen zu lassen.

7 Glossar

Apache Tapestry	Webframework für die Programmiersprache Java
Hibernate	(englisch für Winterschlaf halten) ist ein Open-Source-Persistenz- und ORM-Framework für Java.
Xpand	Xpand ist eine statisch typisierte Templatesprache mit speziellen, für die Codegenerierung wichtigen Features.
Eclipse (IDE)	Software-Plattform und Integrierte Entwicklungsumgebung
Unified Modeling Language, UML	(Vereinheitlichte Modellierungssprache), kurz UML, ist eine graphische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Software-Teilen und anderen Systemen
M2T	ModelToText, bezeichnet die Generierung von Textdokumenten (Quellcode) aus Modelldateien (UML)
ORM	Objektrelationales Mapping, bildet eine Klassenhierarchie auf ein relationales Tabellenschema ab.
JUnit	Framework zum Testen von Java-Programmen.
MDA	Model-Driven-(Software)-Architecture, modellgetriebene Softwarearchitektur
HSQLDB	Hyper Structured Query Language Database